

15-213

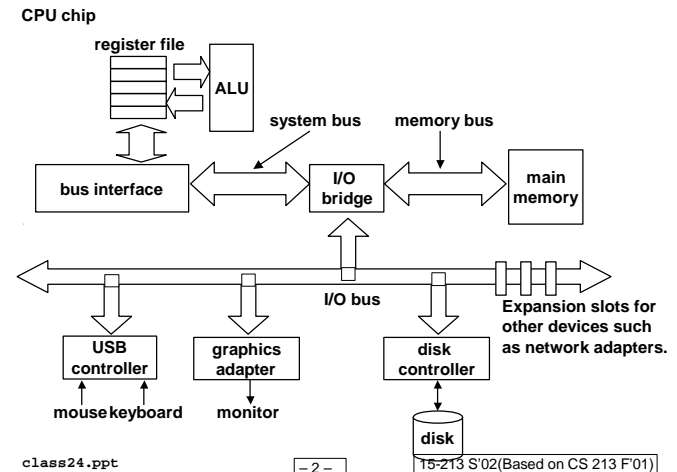
I/O
April 16, 2002

Topics

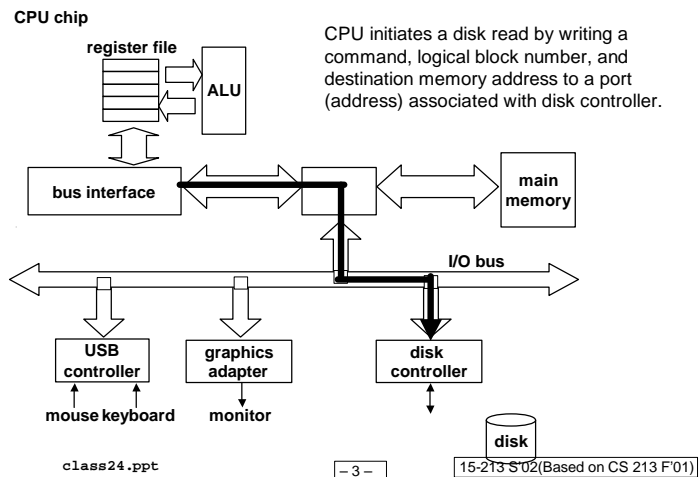
- Files
- Unix I/O
- Standard I/O
- Reading: 12.1-12.4 (Beta) or 12.1-10.1 (New)
- Problems: 12.9 (Beta) or 12.4 (New)

class24.ppt

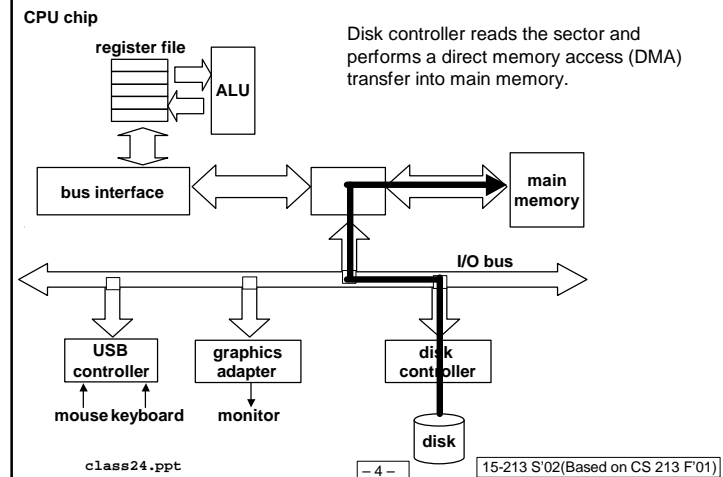
A typical hardware system



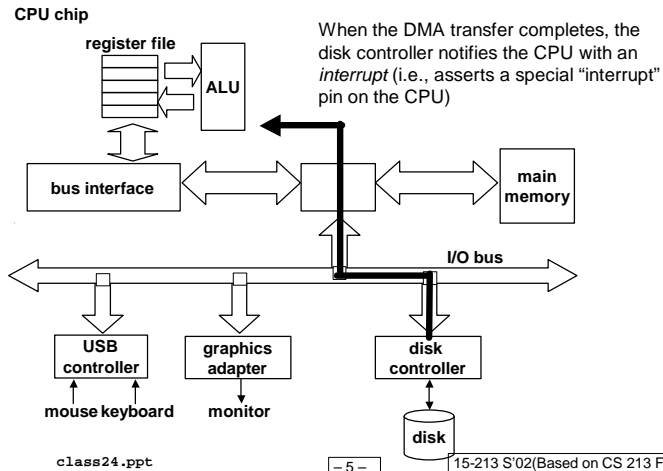
Reading a disk sector (1)



Reading a disk sector (2)



Reading a disk sector (3)



Unix files

A Unix file is a sequence of n bytes:

- $B_0, B_1, \dots, B_k, \dots, B_{n-1}$

All I/O devices are represented as files:

- `/dev/sda2` (/usr disk partition)
- `/dev/tty2` (terminal)

Even the kernel is represented as a file:

- `/dev/kmem` (kernel memory image)
- `/proc` (kernel data structures)

Kernel identifies each file by a small integer *descriptor*:

- 0: standard input
- 1: standard output
- 2: standard error

class24.ppt

- 6 -

15-213 S'02(Based on CS 213 F'01)

Unix file types

Regular file

- Binary or text file.
- Unix does not know the difference!

Directory file

- A file that contains names and locations of other files.

Character special file

- For certain types of streaming devices (e.g., terminals)

Block special file

- A file type typically used for disks.

FIFO (named pipe)

- A file type used for interprocess communication

Socket

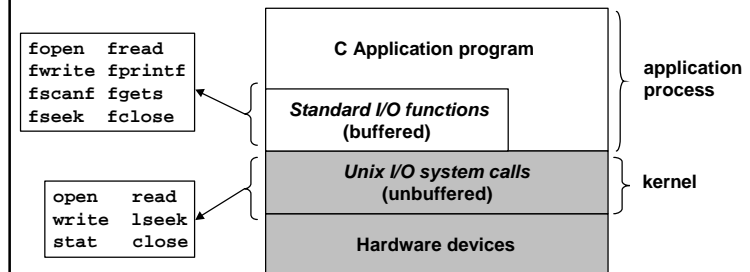
- A file type used for network communication between processes

class24.ppt

- 7 -

15-213 S'02(Based on CS 213 F'01)

I/O functions



class24.ppt

- 8 -

15-213 S'02(Based on CS 213 F'01)

Standard I/O vs Unix I/O

When should I use the Standard I/O functions?

- Whenever possible!
- Many C programmers are able to do all of their work using the standard I/O functions.

Why would I ever need to use the lower level Unix I/O functions?

- Fetching file metadata (e.g., size, modification date)
 - must use lower level stat function
- Doing network programming
 - there are some bad interactions between sockets and the standard library functions.
- Needing high performance.

class24.ppt

– 9 –

15-213 S'02(Based on CS 213 F'01)

Meta-data returned by stat ()

```
/*
 * file info returned by the stat function
 */
struct stat {
    dev_t      st_dev;      /* device */
    ino_t      st_ino;      /* inode */
    mode_t     st_mode;     /* protection and file type */
    nlink_t    st_nlink;    /* number of hard links */
    uid_t      st_uid;      /* user ID of owner */
    gid_t      st_gid;      /* group ID of owner */
    dev_t      st_rdev;     /* device type (if inode device) */
    off_t      st_size;     /* total size, in bytes */
    unsigned long st_blksize; /* blocksize for filesystem I/O */
    unsigned long st_blocks; /* number of blocks allocated */
    time_t     st_atime;    /* time of last access */
    time_t     st_mtime;    /* time of last modification */
    time_t     st_ctime;    /* time of last change */
};
```

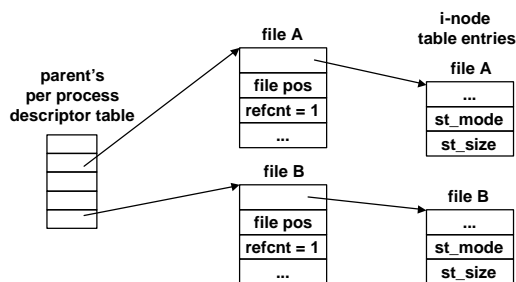
class24.ppt

– 10 –

15-213 S'02(Based on CS 213 F'01)

Kernel's file representation

Before fork:



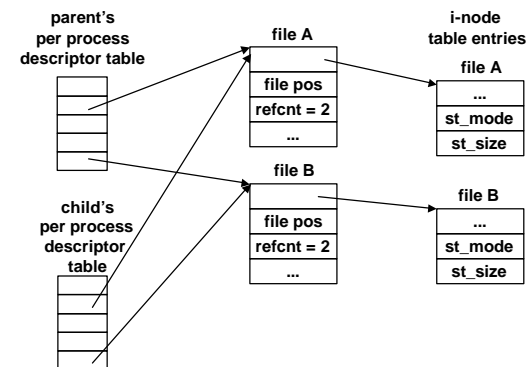
class24.ppt

– 11 –

15-213 S'02(Based on CS 213 F'01)

Kernel's file representation (cont)

After fork (processes inherit open file tables):



class24.ppt

– 12 –

15-213 S'02(Based on CS 213 F'01)

Unix file I/O: open ()

Must open () a file before you can do anything else.

```
int fd; /* file descriptor */

if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {
    perror("open");
    exit(1);
}
```

open () returns a small integer (file descriptor)

- fd < 0 indicates that an error occurred

predefined file descriptors:

- 0: stdin
- 1: stdout
- 2: stderr

class24.ppt

- 13 -

15-213 S'02(Based on CS 213 F'01)

Unix file I/O: read ()

read () allows a program to fetch the contents of a file from the current *file position*.

```
char buf[512];
int fd; /* file descriptor */
int nbytes; /* number of bytes read */

/* open the file */
...
/* read up to 512 bytes from file fd */
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {
    perror("read");
    exit(1);
}
```

read () returns the number of bytes read from file fd.

- nbytes < 0 indicates that an error occurred.
- short counts (nbytes < sizeof(buf)) are possible and not errors!
- places nbytes bytes into memory address buf

class24.ppt

- 14 -

15-213 S'02(Based on CS 213 F'01)

Unix file I/O: write ()

write () allows a program to modify the file contents starting at the current file position.

```
char buf[512];
int fd; /* file descriptor */
int nbytes; /* number of bytes read */

/* open the file */
...
/* write up to 512 bytes from buf to file fd */
if ((nbytes = write(fd, buf, sizeof(buf))) < 0) {
    perror("write");
    exit(1);
}
```

write () returns the number of bytes written from buf to file fd.

- nbytes < 0 indicates that an error occurred.
- short counts are possible and are not errors.

class24.ppt

- 15 -

15-213 S'02(Based on CS 213 F'01)

Robustly dealing with read short counts (adapted from Stevens)

```
ssize_t readn(int fd, void *buf, size_t count)
{
    size_t nleft = count;
    ssize_t nread;
    char *ptr = buf;
    while (nleft > 0) {
        if ((nread = read(fd, ptr, nleft)) < 0) {
            if (errno == EINTR)
                nread = 0; /* and call read() again */
            else
                return -1; /* errno set by read() */
        }
        else if (nread == 0)
            break; /* EOF */
        nleft -= nread;
        ptr += nread;
    }
    return (count - nleft); /* return >= 0 */
}
```

class24.ppt

- 16 -

15-213 S'02(Based on CS 213 F'01)

Robustly dealing with write short counts (adapted from Stevens)

```
ssize_t writen(int fd, const void *buf, size_t count)
{
    size_t nleft = count;
    ssize_t nwritten;
    const char *ptr = buf;

    while (nleft > 0) {
        if ((nwritten = write(fd, ptr, nleft)) <= 0) {
            if (errno == EINTR)
                nwritten = 0; /* and call write() again */
            else
                return -1; /* errno set by write() */
        }
        nleft -= nwritten;
        ptr += nwritten;
    }
    return count;
}
```

class24.ppt

- 17 -

15-213 S'02(Based on CS 213 F'01)

Unix file I/O: lseek()

lseek() changes the current file position

```
int fd; /* file descriptor */

/* open the file */
...
/* Move the file position to byte offset 100 */
if ((lseek(fd, 100, SEEK_SET) < 0) {
    perror("lseek");
    exit(1);
}
```

lseek() changes the current file position.

class24.ppt

- 18 -

15-213 S'02(Based on CS 213 F'01)

Unix file I/O: dup2()

dup2() copies entries in the per-process file descriptor table.

```
/* Redirect stderr to stdout (so that driver will get all output
on the pipe connected to stdout) */
dup2(1, 2); /* copies fd 1 (stdout) to fd 2 (stderr) */
```

lseek(oldfd, newfd) overwrites the entry in the per-process file table for newfd with the entry for oldfd.

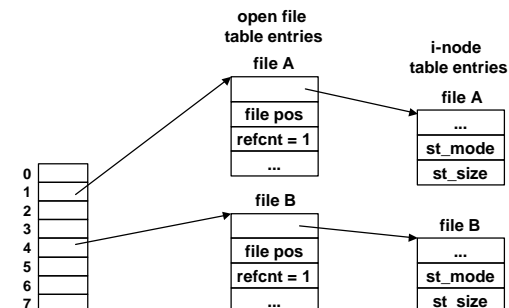
class24.ppt

- 19 -

15-213 S'02(Based on CS 213 F'01)

dup2() example

Before calling dup2(4, 1) :



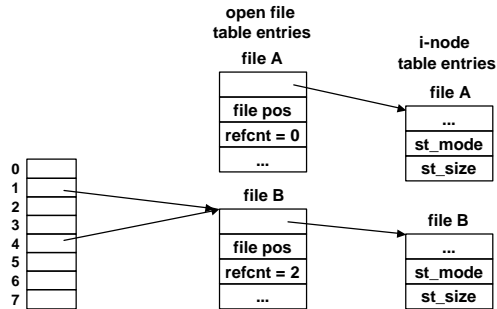
class24.ppt

- 20 -

15-213 S'02(Based on CS 213 F'01)

dup2() example

After calling dup2(4, 1) :



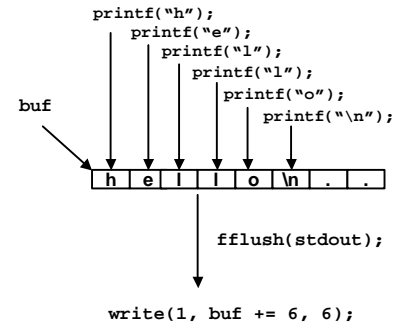
class24.ppt

- 21 -

15-213 S'02(Based on CS 213 F'01)

Buffering in Standard I/O

Standard library functions use *buffered* I/O



class24.ppt

- 22 -

15-213 S'02(Based on CS 213 F'01)

Buffering in action

You can see this buffering for yourself, using the always fascinating Unix strace program:

```
#include <stdio.h>

int main()
{
    printf("h");
    printf("e");
    printf("l");
    printf("l");
    printf("o");
    printf("\n");
    fflush(stdout);
    exit(0);
}
```

```
bass> strace hello
execve("./hello", ["hello"], [/ * ... */]).
...
write(1, "hello\n", 6)           = 6
munmap(0x40000000, 4096)         = 0
_exit(0)                         = ?
```

class24.ppt

- 23 -

15-213 S'02(Based on CS 213 F'01)

Using buffering to robustly read text lines (Stevens)

```
static ssize_t my_read(int fd, char *ptr)
{
    static int read_cnt = 0;
    static char *read_ptr, read_buf[MAXLINE];

    if (read_cnt <= 0) {
        again:
        if ( (read_cnt = read(fd, read_buf, sizeof(read_buf))) < 0) {
            if (errno == EINTR)
                goto again;
            return -1;
        }
        else if (read_cnt == 0)
            return 0;
        read_ptr = read_buf;
    }
    read_cnt--;
    *ptr = *read_ptr++;
    return 1;
}
```

class24.ppt

- 24 -

15-213 S'02(Based on CS 213 F'01)

Robustly reading text lines (cont)

```
ssize_t readline(int fd, void *buf, size_t maxlen) {
    int n, rc;
    char c, *ptr = buf;
    for (n = 1; n < maxlen; n++) { /* notice that loop starts at 1 */
        if ( (rc = my_read(fd, &c)) == 1) {
            *ptr++ = c;
            if (c == '\n')
                break; /* newline is stored, like fgets() */
        }
        else if (rc == 0) {
            if (n == 1)
                return 0; /* EOF, no data read */
            else
                break; /* EOF, some data was read */
        }
        else
            return -1; /* error, errno set by read() */
    }
    *ptr = 0; /* null terminate like fgets() */
    return n;
}
```

class24.ppt

- 25 -

15-213 S'02(Based on CS 213 F'01)

mmap() revisited

```
void *mmap(void *start, int len, int prot,
           int flags, int fd, int offset)
```

Map **len** bytes starting at offset **offset** of the file specified by file description **fd**, preferably at address **start** (usually 0 for don't care).

- **prot**: MAP_READ, MAP_WRITE
- **flags**: MAP_PRIVATE, MAP_SHARED

Return a pointer to the mapped area

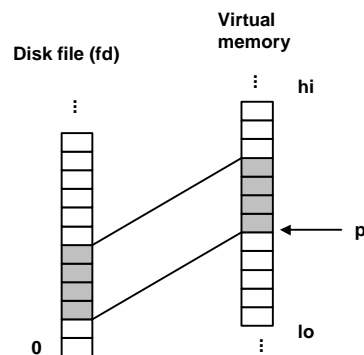
class24.ppt

- 26 -

15-213 S'02(Based on CS 213 F'01)

Visualizing mmap()

```
p = mmap(NULL, 4, PROT_READ, MAP_PRIVATE, fd, 2);
```



class24.ppt

- 27 -

15-213 S'02(Based on CS 213 F'01)

mmap() example

```
/* mmapcopy - uses mmap to copy file fd to stdout */
void mmapcopy(int fd, int size)
{
    char *bufp; /* ptr to memory mapped VM area */
    bufp = Mmap(NULL, size, PROT_READ, MAP_PRIVATE, fd, 0);
    Write(1, bufp, size);
    return;
}

/* mmapcopy driver */
int main(int argc, char **argv)
{
    struct stat stat;
    int fd;

    fd = Open(argv[1], O_RDONLY, 0);
    Fstat(fd, &stat);
    mmapcopy(fd, stat.st_size);
    exit(0);
}
```

class24.ppt

- 28 -

15-213 S'02(Based on CS 213 F'01)

For further information

The Unix bible:

- W. Richard Stevens, *Advanced Programming in the Unix Environment*, Addison Wesley, 1993.

Stevens is arguably the best technical writer ever.

Produced authoritative works in:

- Unix programming
- TCP/IP (the protocol that makes the Internet work)
- Unix network programming
- Unix IPC programming.

Tragically, Stevens died Sept 1, 1999.